Quasi-Static Scheduling of Communicating Tasks

Philippe Darondeau [a,1]

[a] *IRISA/INRIA, Campus de Beaulieu, Rennes, France*

Blaise Genest, [b,*,1,2]

[b] *IRISA/CNRS, Campus de Beaulieu, Rennes, France*

P.S. Thiagarajan [c]

[c] *School of Computing, National University of Singapore*

Shaofa Yang [d,1]

[d] *UNU-IIST, Macao* [3]

Abstract

Good scheduling policies for distributed embedded applications are required for meeting hard real time constraints and for optimizing the use of computational resources. We study the *quasi-static scheduling* problem in which (uncontrollable) control flow branchings can influence scheduling decisions at run time. Our abstracted distributed task model consists of a network of sequential processes that communicate via point-to-point buffers. In each round, the task gets activated by a request from the environment. When the task has finished computing the required responses, it reaches a pre-determined configuration and is ready to receive a new request from the environment. For such systems, we prove that determining the existence of a scheduling policy that guarantees upper bounds on buffer capacities is undecidable. However, we show that the problem is decidable for the important subclass of "data-branching" systems in which control flow branchings are exclusively due to data-dependent internal choices made by the sequential components. This decidability result exploits ideas derived from the Karp and Miller coverability tree for Petri nets as well as the existential boundedness notion of languages of message sequence charts.

Key words: Communicating machines, Quasi-static scheduling, Channel bound.
1991 MSC: 68N30

1 Introduction

The high complexity of embedded systems poses challenges for their design and verification. To tame the complexity, a possible design methodology is to use specifications that are intrinsically concurrent and asynchronous, such as data flow networks [2], Kahn process networks [9], and Petri nets [15]. To implement a large system of interactive tasks on a collection of hardware resources, one partitions the large specification into small clusters of processes and one subsequently implements each cluster independently, see e.g. [3]. In specifications, processes communicate via buffers, that are allowed to be arbitrarily large. Clearly, in implementations, one must use a finite amount of resources, and in particular, a finite capacity for communication buffers inside each cluster. A basic problem is thus how to schedule processes properly within each cluster, considered as a separate system, so that asynchronous buffers internal to the cluster never exceed some finite bound. We model each cluster as a finite system of processes communicating via point-to-point buffers. Each process is a sequential transition system, in which non-deterministic branchings may have two origins: (i) a data-dependent internal choice made by a sequential component; (ii) a process waiting for messages on different input buffers. In the second case, the waiting process non-deterministically branches by picking up a message from one of the *nonempty* input buffers [4]. The system of processes is triggered by the environment iteratively in *rounds*. We model the system dynamics for just one round. It is easy to lift results to multiple rounds. In each round, the environment sends a data item to one of the processes. This communication starts the computation to be done in the round. The computation finishes successfully when all processes are in their final states and all buffers are empty. Then, the system waits for the initialization of a new round by the environment. In a technical sense, buffers—which are viewed here as counters without zero tests—are deployed as over-approximations of FIFOs whereas, using FIFOs directly would make the model Turing powerful [1]. In the present setting, we are interested in determining a good schedule for the processes: If at some configuration the scheduler picks the process p for execution and p is at a state with several outgoing transitions, then we require that a good schedule allows *all* possible choices to occur. In the sequel, such schedules are referred to as *quasi-static* schedules. In addition, a good schedule should never prevent the system from

* Corresponding author. Address: IRISA, Campus de Beaulieu, Rennes, France.
Email addresses: darondeau@irisa.fr (Philippe Darondeau), bgenest@irisa.fr (Blaise Genest,), thiagu@comp.nus.edu.sg (P.S. Thiagarajan), ysf@iist.unu.edu (Shaofa Yang).
[1] Work done as part of the Associated Team DST.
[2] Work supported by the ANR-SETI-06 DOTS.
[3] Work done while being affiliated with IRISA/INRIA and supported by CREATE ACTIVEDOC.

(eventually) reaching the final state. Schedules with this property are called here *valid* schedules. Finally, a good schedule is required to be *regular* in the sense that the system under schedule should use only a *bounded* amount of memory for serving the request made by the environment. In particular, the schedule should enforce a uniform upper bound on the number of items stored in the buffers during the round.

We show first that it is undecidable whether a valid and regular quasi-static schedule exists. The undecidability result holds even if the system on its own is valid in that it is possible to reach the final global state from every reachable global state of the unscheduled system. Next we define the subclass of data-branching systems in which the simultaneous polling on multiple input buffers is ruled out; hence the only branching allowed is local (data-dependent) branching. We show that for data-branching systems, one can effectively check whether there exists a valid and regular quasi-static schedule. This result is obtained by applying classical ideas from [10] to a special scheduling policy that we define, called the canonical schedule. The canonical schedule is based on the same ideas as the normal form used for the existential boundedness of languages of message sequence charts [7]. The crucial point is that one cannot directly apply the techniques of [10] to the scheduling problem, because the canonical schedule uses zero tests on buffers. It is well known that zero tests often lead to undecidability, but fortunately this is not the case here.

Before reviewing related work, it is worth noting that our setting is strongly oriented towards round-based executions of distributed tasks. Hence it does not cater for models capturing non-terminating computations such as Kahn process networks [9]. It is not clear at present whether our undecidability result can be extended to such settings. Quasi-static scheduling (QSS) has been studied in the past in a number of settings (see [11] for a survey). The early work [2] studied dynamic scheduling of boolean-controlled dataflow graphs. As this computation model is Turing powerful, the QSS problem is undecidable for this class of systems [2]. Later, [4] proposed a heuristic to solve the QSS problem on a different model called the YAPI model by exploring only a subset of the infinite state space. There is however no proof that the heuristic is complete, even for a subset of YAPI models. The work [12] considered the QSS problem on a restricted class of Petri nets called Equal-Conflict Petri nets and showed the decidability of this problem. However the notion of schedulability used in [12] is much weaker than the one proposed in [4] or the one which we propose here. Basically, under the scheduling regime defined in [12], only a finite number of runs can arise, hence systems with loops are not schedulable. In comparison, the system models which we consider are very close to (general) Petri Nets. Our scheduling notion is essentially the same as in [4], but slightly modified to fit our model. Our undecidability result is harder to obtain than the similar result in [2], since reachability is decidable for our system models. Indeed, the decidability of our quasi-static schedulability problem is stated as

an open problem in [4,11]. The work [15] considered QSS in the setting of [4] and proposed a sufficient (but not necessary) condition for non-schedulability based on the structure of the Petri net system model. There is also previous research concerning FIFOs, proposing a semi-effective check for schedulability [16] as well as a necessary condition that implies non-schedulability [8]. These cited works use methods similar to ours and [10]. However these results do no establish clear-cut boundaries between the decidable and undecidable partly due to the expressiveness of unbounded FIFOs.

In the next section we present our model of systems and the quasi-static scheduling problem. Section 3 establishes the undecidability result in the general setting. Section 4 imposes the data-branching restriction and shows the decidability of the quasi-static scheduling problem under this restriction. The final section summarizes and discusses the results. This paper is a complete version of the extended abstract [5].

2 Preliminaries

Through the rest of the paper, we fix a finite set $\mathcal{P}$ of process names. Accordingly, we fix a finite set Ch of buffer names. To each buffer c, we associate a source process and a destination process, denoted $src(c)$ and $dst(c)$ respectively. We have $src(c) \neq dst(c)$ for each $c \in Ch$. For each process p, we set $\Sigma_p^! = \{!c \mid c \in Ch, src(c) = p\}$ and $\Sigma_p^? = \{?c \mid c \in Ch, dst(c) = p\}$. So, $!c$ stands for the action that deposits one item into the buffer c while $?c$ is the action that removes one item from c. For each p, we fix also a finite set Σ_p^{cho} of choice actions. We assume that $\Sigma_p^{cho} \cap \Sigma_q^{cho} = \emptyset$ whenever $p \neq q$. Members of Σ_p^{cho} will be used to label branches arising from the abstraction of data dependences in the "if...then...else", "switch..." and "while..." statements executed by the process p. For each p, we set $\Sigma_p = \Sigma_p^! \cup \Sigma_p^? \cup \Sigma_p^{cho}$. Note that $\Sigma_p \cap \Sigma_q = \emptyset$ whenever $p \neq q$. Finally, we fix $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$.

A *task system* (abbreviated as "system" from now on) is a structure $\mathcal{A} = \{(S_p, s_p^{init}, \longrightarrow_p, s_p^{fi})\}_{p \in \mathcal{P}}$, where for each $p \in \mathcal{P}$, S_p is a finite set of states, s_p^{init} is the initial state, $\longrightarrow_p \subseteq S_p \times \Sigma_p \times S_p$ is the transition relation, and s_p^{fi} is the final state. As usual, if $s_p \in S_p$ and $\delta = (\hat{s}_p, a_p, \hat{s}'_p)$ is in $\longrightarrow_p$ with $\hat{s}_p = s_p$, then we call δ an outgoing transition of s_p. We require the following conditions to be satisfied:

- For each $p \in \mathcal{P}$ and $s_p \in S_p$, if the set of outgoing transitions of s_p is not empty, then exactly one of the following conditions holds:
  - Every outgoing transition of s_p is in $S_p \times \Sigma^{cho} \times S_p$. Such a state s_p is a *(data-dependent) choice* state.

- s_p has exactly one outgoing transition and this transition is a send $(s_p, !c, s'_p)$, where $c \in Ch, s'_p \in S_p$. Such a state s_p is a *sending* state.
- Every outgoing transition of s_p is in $S_p \times \Sigma_p^? \times S_p$. Such a state s_p is a *polling* state.

- For each process p, the final state s_p^{fi} either has no outgoing transitions or it is a polling state.

The system works in rounds. When the first round starts, all the processes will be in their initial states and the buffers will be empty (it is easy to lift the results in the paper to any other initialization of the buffers, modeling different memory states). The first round starts when a message from the environment is received on a designated channel by a designated process (the same for each round). At the end of each round, every process will be in its final state, and all buffers will be empty. A reset operation —possibly triggered by the environment—is assumed to be performed to initiate a new round. This operation puts every process in its initial state from which the computation can start again (upon receiving a message from the environment as described above). Thus, computations belonging to different rounds will not get mixed up. We do not explicitly represent this reset operation in the system model. For technical convenience, we do not consider multi-rate communications where multiple items can be deposited to or picked up from a buffer at one time. However, our results extend easily to multi-rate task systems. They can also be adapted to systems where several rounds can overlap (e.g. pipelines), as long as the system terminates (this rules out general Kahn process networks).

For notational convenience, we shall assume that the system is *deterministic*, that is for each p, for each $s_p \in S_p$, if $(s_p, a1, s1_p)$, $(s_p, a2, s2_p)$ are in $\longrightarrow_p$, then $a1 = a2$ implies $s1_p = s2_p$. However, all our results can be extended easily to non-deterministic systems. The dynamics of a system $\mathcal{A}$ are defined by the transition system $TS_{\mathcal{A}}$ which we describe now. A configuration of $\mathcal{A}$ is a pair (s, χ) where $s \in \prod_{p \in \mathcal{P}} S_p$ and χ is a mapping that assigns to each buffer c in Ch a non-negative integer $\chi(c)$ indicating the number of items it contains. We term the members of $\prod_{p \in \mathcal{P}} S_p$ as *global* states. We view a global state as a mapping from $\mathcal{P}$ to $\bigcup_{p \in \mathcal{P}} S_p$ such that $s(p) \in S_p$ for each p. When no confusion arises, we write s_p for $s(p)$. The *initial* configuration of $\mathcal{A}$ is (s^{init}, χ^0) where $s^{init}(p) = s_p^{init}$ for each p, and $\chi^0(c) = 0$ for every $c \in Ch$. We define $TS_{\mathcal{A}} = (RC_{\mathcal{A}}, (s^{init}, \chi^0), \Longrightarrow_{\mathcal{A}})$ where the (possibly infinite) set $RC_{\mathcal{A}}$ of reachable configurations and the global transition relation $\Longrightarrow_{\mathcal{A}} \subseteq RC_{\mathcal{A}} \times \Sigma \times RC_{\mathcal{A}}$ are the least sets satisfying the following:

- $(s^{init}, \chi^0) \in RC_{\mathcal{A}}$.
- Suppose (s, χ) is in $RC_{\mathcal{A}}$ and let $(s(p), a, s'_p) \in \longrightarrow_p$ such that $a = ?c$ entails $\chi(c) \geq 1$. Then $(s', \chi') \in RC_{\mathcal{A}}$ and $((s, \chi), a, (s', \chi')) \in \Longrightarrow_{\mathcal{A}}$, where $s'(p) = s'_p$, $s'(q) = s(q)$ for all $q \neq p$, and χ' is the map defined as follows:

- If $a = !c$, then $\chi'(c) = \chi(c) + 1$ and $\chi'(d) = \chi(d)$ for all $d \neq c$.
- If $a = ?c$, then $\chi'(c) = \chi(c) - 1$ and $\chi'(d) = \chi(d)$ for all $d \neq c$.
- If $a \in \Sigma_p^{cho}$, then $\chi'(c) = \chi(c)$ for all $c \in Ch$.

We define s^{fi} as the global state given by $s^{fi}(p) = s_p^{fi}$ for each p. We term (s^{fi}, χ^0) as the *final* configuration.

For a sequence $\sigma = a_1 \cdots a_{n-1} \in \Sigma^*$ and two configurations (s, χ) and (s', χ'), we write $(s, \chi) \stackrel{\sigma}{\Longrightarrow} (s', \chi')$ whenever there exist $(s_i, \chi_i)_{1 \leq i \leq n}$ with $(s_1, \chi_1) = (s, \chi)$, $(s_n, \chi_n) = (s', \chi')$ and for all $1 \leq i < n$, $(s_i, \chi_i) \stackrel{a_i}{\Longrightarrow} (s_{i+1}, \chi_{i+1})$. We define a *run* of $\mathcal{A}$ as a sequence $\sigma \in \Sigma^*$ such that $(s^{init}, \chi^0) \stackrel{\sigma}{\Longrightarrow} (s, \chi)$ for some (s, χ) in $RC_{\mathcal{A}}$. We say that σ *ends at* configuration (s, χ), and denote this configuration by (s^σ, χ^σ). We let $Run(\mathcal{A})$ denote the set of runs of $\mathcal{A}$. The run σ is *complete* iff $(s^\sigma, \chi^\sigma) = (s^{fi}, \chi^0)$, and we denote by $Run_{cpl}(\mathcal{A})$ the set of complete runs of $\mathcal{A}$.

Through the rest of this section, we fix a system $\mathcal{A}$ and we therefore write RC and Run_{cpl} instead of $RC_{\mathcal{A}}$ and $Run_{cpl}(\mathcal{A})$. A configuration (s, χ) in RC is *valid* iff there exists σ such that $(s, \chi) \stackrel{\sigma}{\Longrightarrow} (s^{fi}, \chi^0)$. A run σ is *valid* iff it ends at a valid configuration. We say that $\mathcal{A}$ is *deadend-free* iff every configuration in RC is valid. Note that one can effectively decide whether a given system $\mathcal{A}$ is deadend-free by an easy reduction to the home marking reachability problem of Petri nets [6].

To illustrate the main concepts, we give an example taken from [11], slightly modified to fit our context. We consider a cluster consisting of two processes shown in Fig. 1 and Fig. 2. The two processes communicate through the FIFO channel `Port`. The cluster communicates with the environment, by reading n from channel `Start`, and by executing the commands `GetData()` and `SendData()`. Notice that there is no uniform bound on runs of this system: for all B, there is a run starting by `read(Start,`$n = B + 1$`)` and with $B + 1$ messages (in transit) in channel `Port`.

We abstract each round of this cluster as the system of communicating processes shown in Fig.3. $P1$ is started by receiving from the environment an integer n which it reads from buffer `Start`, and $P2$ is started by receiving

```
while(true)
  { read(Start,n);
    for i = 1 to n
      { GetData(x);
        send(Port,x^2); }
    send(Port,"end");
  }
```

Fig. 1. *Process 1 of the cluster*

```
while(true)
  { read(Port,z); y ← 0;
    while(z ≠ "end")
      { y ← y + z;
        read(Port,z); }
    SendData(y);
  }
```

Fig. 2. *Process 2 of the cluster*

from $P1$ a message which it reads from buffer `Port`. We abstract away the other communications with the environment. We also abstract away the data x, y, z and n, and their arithmetic handling. The `for` loop is then replaced with a cycle. Since we do not have FIFO channels (as they would make our model Turing complete), we replace channel `Port` with two buffers d (counting messages with a data content) and e (counting messages with content `"end"`). The initial states are A and 1 while D and 2 are final states. The sequence `endfor` $!e\,?e$ is a complete run. The run $\sigma =$ `for` $!d$ `endfor` $!e\,?e$ is not complete, even though $s^\sigma = (D, 2)$. For, we have $\chi^\sigma(d) = 1 \neq 0$. This system is not deadend-free, since the run σ cannot be extended to a complete run.

2.1 Schedules

We now define the notion of schedules and schedulability. Let $(s, \chi) \in RC$ be a reachable configuration of $\mathcal{A}$. The action $a \in \Sigma$ is enabled at (s, χ) iff $(s, \chi) \stackrel{a}{\Longrightarrow} (s', \chi')$ for some (s', χ') in RC. On the other hand, the process $p \in \mathcal{P}$ is enabled at (s, χ) iff some $a \in \Sigma_p$ is enabled at (s, χ). A *schedule* for $\mathcal{A}$ is a partial function Sch from Run to $\mathcal{P}$ which satisfies the following conditions: $Sch(\sigma)$ is defined iff some action a is enabled at (s^σ, χ^σ), and if $Sch(\sigma) = p$, then p is enabled at (s^σ, χ^σ). Notice that if σ is a complete run, then no action is enabled at (s^σ, χ^σ) and $Sch(\sigma)$ is therefore undefined (in notation, $Sch(\sigma) = \epsilon$). Given a schedule Sch, we denote by Run/Sch the set of runs *in agreement with* Sch and we define this set inductively as follows: first, the empty sequence $\varepsilon \in Run/Sch$; second, if $\sigma \in Run/Sch$, $Sch(\sigma) = p$, $a \in \Sigma_p$ and σa is a run, then $\sigma a \in Run/Sch$. In particular, if $Sch(\sigma) = p$ and σ can be extended by two alternative actions a, b of process p, then the schedule *must* allow both a and b. It is easy to check that this definition of a schedule is equivalent to the one given in [4].

The schedule Sch is *valid* iff every run in Run/Sch can be extended to a run in $Run/Sch \cap Run_{cpl}$. Next we define $RC/Sch = \{(s^\sigma, \chi^\sigma) \mid \sigma \in Run/Sch\}$, the set of configurations reached via runs in agreement with Sch. We say that Sch is *regular* if RC/Sch is a finite set and Run/Sch is a regular language (in particular, the system under schedule can be implemented with finite memory). Finally, we say that $\mathcal{A}$ is *quasi-static schedulable* (schedulable for short) iff there exists a *valid and regular* schedule for $\mathcal{A}$. The quasi-static scheduling

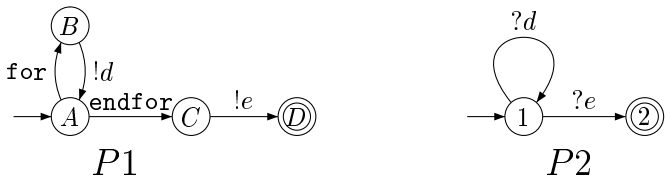

Fig. 3. A task system with two processes $P1, P2$.

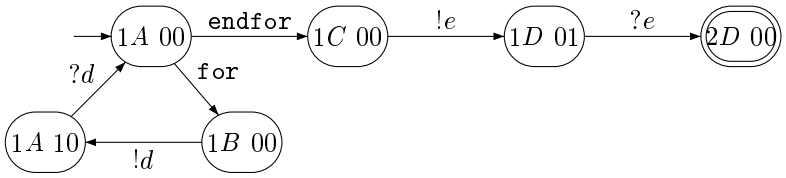

Fig. 4. The system under schedule RC/Sch_2.

problem is to determine, given a system $\mathcal{A}$, whether $\mathcal{A}$ is schedulable. Again, it is easy to check that this definition of quasi-static schedulability is equivalent to the one given in [4]. In particular, the validity of the schedule corresponds to the requirement, made in [4], that the system can always answer a query of the environment (which is guaranteed here by reaching the final configuration).

Let us consider again the task system with two processes $P1, P2$ and two channels c, d presented earlier in this section and shown in Fig. 3. The function $Sch_1(\sigma) = P$ defined with $P = P1$ if $P1$ is enabled at state (s^σ, χ^σ), $P = P2$ otherwise, is a schedule. However, this schedule is not regular, since for all m, $(\texttt{for}\,!d)^m \in Run/Sch_1$ reaches a configuration with m messages sent in the channel d, implying that this channel is not bounded. On the other hand, the function $Sch_2(\sigma) = P$ defined with $P = P2$ if $P2$ is enabled at state (s^σ, χ^σ), $P = P1$ otherwise is a valid and regular schedule. Fig. 4 shows the finite state space RC/Sch_2 which contains no deadends. In this figure, a configuration is described in the form $XY\,\alpha\beta$, where X (Y) is the state of $P2$ $(P1)$, and α, β denote the contents of buffer d, e respectively. Thus the system of Fig. 3 is schedulable. Notice that a valid schedule does not need to prevent infinite runs. It just must *allow* every run to be completed for some sequence of inputs and / or choices.

3 General Case and Undecidability

The goal of this section is to establish the following result.

Theorem 1 *The quasi-static scheduling problem is undecidable. Moreover, this problem remains undecidable for the sub-class of systems without deadends.*

Our proof consists in showing that the halting problem for deterministic two-counter machines can be uniformly reduced to the quasi-static scheduling problem. Specifically, given a deterministic two counter machine $\mathcal{M}$, we shall construct a system $\mathcal{A}$ such that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable. Following a standard technique used by numerous authors, some runs of $\mathcal{A}$ will not correspond to any run of $\mathcal{M}$. Hence $\mathcal{A}$ will not faithfully simulate $\mathcal{M}$. However such runs will be unbounded or lead to deadends, and these runs shall be avoided by

the scheduler.

To ease the understanding, we shall present the construction of $\mathcal{A}$ in three phases and prove in each case that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable. In the first phase, our goal is to bring out the main ingredients of the construction of $\mathcal{A}$ with a minimal amount of technical details. Thus, we shall allow transitions of $\mathcal{A}$ to deviate from the definition of systems given in Section 2. In the second phase, we modify the transitions of $\mathcal{A}$ given in the first phase, so that they strictly adhere to the definition of systems in Section 2. In the first and second phase, $\mathcal{A}$ needs not be deadend-free. In the third phase, we show that the system $\mathcal{A}$ constructed in the second phase can be modified to become deadend-free while strictly adhering to the definition of systems in Section 2, establishing thus the second part of the theorem.

In the first two phases $\mathcal{A}$ will enjoy the following two properties: first, if Sch is a valid schedule for $\mathcal{A}$, then the execution of $\mathcal{A}$ under the schedule Sch simulates the execution of $\mathcal{M}$; second, if Sch leads $\mathcal{A}$ to its final configuration, then the corresponding execution of $\mathcal{M}$ reaches the halting state. We will show that whenever $\mathcal{M}$ halts, there exists a valid schedule Sch that leads $\mathcal{A}$ to its final configuration in a finite number of steps, hence it is a valid and regular schedule and $\mathcal{A}$ turns out to be schedulable. We will show on the other hand that, if $\mathcal{M}$ does not halt, then $\mathcal{A}$ does not even have a valid schedule.

We now give a sketch of the coding of $\mathcal{M}$ by $\mathcal{A}$. Let C_1, C_2 denote the two counters of $\mathcal{M}$. Let *halt* denote the halting state of $\mathcal{M}$. We assume that, for each control state i other than *halt*, the behaviour of $\mathcal{M}$ at i is given by an instruction in one of the following forms, with $j \in \{1, 2\}$:

- $(i, Inc(j), k)$: "increment C_j and move to control state k".
- $(i, Dec(j), k, m)$: "if $C_j > 0$, then decrement C_j and move to control state k; otherwise ($C_j = 0$), move to control state m".

Thus, $\mathcal{M}$ either stops at *halt* after a finite number of steps, or runs forever without visiting *halt*.

Naturally, we encode counters of $\mathcal{M}$ by buffers of $\mathcal{A}$. Incrementing a counter of $\mathcal{M}$ amounts to sending a data item to the corresponding buffer, and decre-

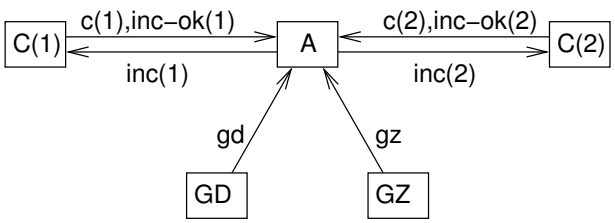

Fig. 5. The architecture of $\mathcal{A}$

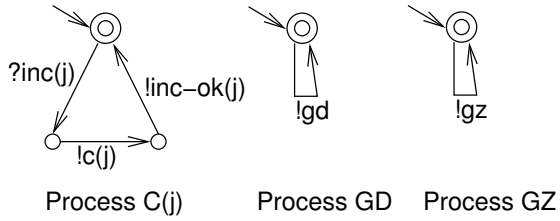

Fig. 6. Description of processes $GD, GZ, C(j)$

menting a counter of $\mathcal{M}$ amounts to picking up a data item from the corresponding buffer. It is clear how the instruction $(i, Inc(j), k)$ of $\mathcal{M}$ can be simulated. The main difficulty is to simulate the instruction $(i, Dec(j), k, m)$. Indeed, in a system, a process can *not* branch to different states according to whether a buffer is empty or not. Further, when a schedule Sch selects a process p to execute, Sch has to allow all transitions of p that are *enabled* at the current state s_p of p. However, the following observation will facilitate the simulation of an $(i, Dec(j), k, m)$ instruction. Suppose s_p is a polling state with two outgoing transitions labelled $?a$, $?b$, where $src(a) \neq src(b)$. If, prior to selecting p, and assuming both buffers a and b are currently empty, Sch can make the buffer a nonempty (for example, by selecting $src(a)$ to send a data item to a) while keeping b empty (for example, by not selecting $src(b)$), then when Sch selects p, only the $?a$ transition is enabled and executed, while the $?b$ transition is ignored.

After these explanations, we enter now the technical part of the proof of Thm. 1.

—**Phase (i)**: Here we construct a system $\mathcal{A}$ whose transitions slightly deviate from the definition of systems in Section 2. In particular, we allow a final state to be not a polling state and permit the outgoing transitions of a local state to be both receive transitions and choice transitions.

The system $\mathcal{A}$ has five processes $A, C(1), C(2), GD, GZ$. Their communication architecture is illustrated in Fig. 5 where a label ch on an arrow from process p to process q represents a buffer ch with $src(ch) = p$ and $dst(ch) = q$. For $j = 1, 2$, the number of items stored in buffer $c(j)$ encodes the value of

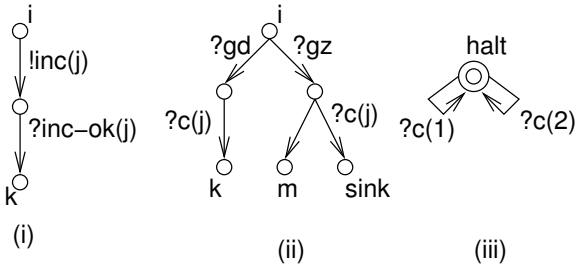

Fig. 7. Transitions of process A

counter C_j of $\mathcal{M}$. Process A mimics the instructions of $\mathcal{M}$. For instructions of the form $(i, Inc(j), k)$, A invokes $C(j)$ to increment $c(j)$ by sending message !inc(j). For instructions of the form $(i, Dec(j), k, m)$, A accepts to receive from both channel gd ("Guess Dec") or gz ("Guess Zero"). The valid schedule can correctly simulate the emptiness test of buffer $c(j)$ by feeding the right channel gd or gz according to the contents of $c(j)$. Fig. 6 displays the transition systems of GD, GZ, and $C(j)$, $j = 1, 2$, where an initial state is indicated by a pointing arrow, and a final state is drawn as a double circle. Fig. 7 illustrates the transition system of A. For each $(i, Inc(j), k)$ instruction of $\mathcal{M}$, A contains the states and transitions shown in Fig. 7(i). For each $(i, Dec(j), k, m)$ instruction of $\mathcal{M}$, A contains the states and transitions shown in Fig. 7(ii), where *sink* is a distinguished state with no outgoing transitions. Unlabelled transitions have implicit labels in Σ^{cho}. For the halting state of $\mathcal{M}$, A contains two special transitions, shown in Fig. 7(iii), whose purpose is to empty the buffers $c(1), c(2)$ after A has reached *halt* for the first time. The initial state of A is the initial state of $\mathcal{M}$, and the final state of A is *halt*.

Let *Sch* be a valid schedule for $\mathcal{A}$. Suppose that, according to *Sch*, the execution of the system $\mathcal{A}$ arrives at a configuration in which process A is at state i. There are two cases to consider:

—**Case (i)**: The corresponding instruction of $\mathcal{M}$ is $(i, Inc(j), k)$.

It is easy to see that *Sch* has no choice but selecting A to execute $!inc(j)$, then selecting $C(j)$ three times in a row to execute $?inc(j), !c(j), !inc\text{-}ok(j)$, and finally selecting A to execute $?inc\text{-}ok(j)$. In doing so, $c(j)$ is incremented and A moves to state k.

—**Case (ii)**: The corresponding instruction of $\mathcal{M}$ is $(i, Dec(j), k, m)$.

Note that from state i of A, there are two outgoing transitions labelled $?gd$, $?gz$ respectively. Consider first the case where $c(j)$ is greater than zero. We argue that *Sch* has to guide A to execute *only* the transition $?gd$ in order to be valid. That is, *Sch* should ensure that the $?gd$ transition of A is enabled by selecting GD. *Sch* must further ensure that the $?gz$ transition of A is *not enabled* which it can do by *not scheduling* the process GZ. By doing so, $c(j)$ will be decremented and A will move to state k. If on the contrary, *Sch* did schedule process GZ and thus enable $?gz$ while $c(j)$ is greater than zero, then *Sch* would allow A to take the $?gz$ transition. Consequently, *Sch* would allow A to reach state m, *as well as* state *sink*. However, as *sink* has no outgoing transitions, the run which leads A to *sink* is not valid. This contradicts the hypothesis that *Sch* is valid. Consider now the case where $c(j)$ is zero. Then it is easy to see that *Sch* has to guide A to execute *only* the transition $?gz$. Further, after executing $?gz$, A can move to state m only, since the corresponding $?c(j)$ transition is not enabled. Altogether, we have shown

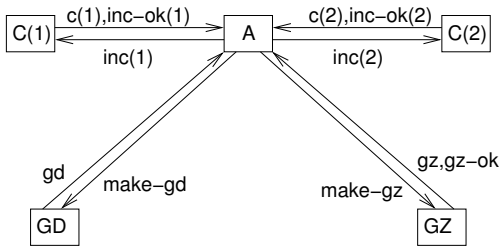

Fig. 8. The architecture of $\mathcal{A}$ in Phase (ii)

that if Sch is a valid schedule, then under this schedule, $\mathcal{A}$ simulates correctly $\mathcal{M}$.

We claim now that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable. To see this, suppose $\mathcal{M}$ halts. Then $\mathcal{M}$ may clearly be simulated by executing $\mathcal{A}$ under some valid schedule Sch that leads $\mathcal{A}$ to the configuration in which each process is at its final state, thus in particular A is in state $halt$, and all buffers except possibly $c(1), c(2)$ are empty. In view of Fig. 7(iii) and the validity of Sch, process A will eventually also empty $c(1), c(2)$. Moreover, it follows also from the finiteness of the run of $\mathcal{A}$ that Sch is regular, hence $\mathcal{A}$ is schedulable. Suppose now that $\mathcal{M}$ does not halt. Assume that Sch is a valid schedule for $\mathcal{A}$. Then as explained above, Sch simulates the execution of $\mathcal{M}$ and thus process A can never reach its final state $halt$. Thus Sch is not valid, a contradiction.

—End of Phase (i)

—**Phase (ii)**: In this phase, we modify the transitions defined for $\mathcal{A}$ in Phase (i) so that they strictly adhere to the definition of system in Section 2.

Firstly, we change the communication architecture between the processes of $\mathcal{A}$. The new architecture is displayed in Fig. 8. The transitions of GD, GZ and processes $C(j)$, $j = 1, 2$, are shown in Fig. 9. Note that the final states of processes GD, GZ are now polling states. For $j = 1, 2$, process $C(j)$ is constructed in the same way as in Phase (i). The state space of A is depicted in Fig. 10. For each $(i, Inc(j), k)$ instruction of $\mathcal{M}$, A contains the transitions shown in Fig. 10(i). For each $(i, Dec(j), k, m)$ instruction of $\mathcal{M}$, A contains the transitions shown in Fig. 10(ii), where $sink$ is a distinguished state with no

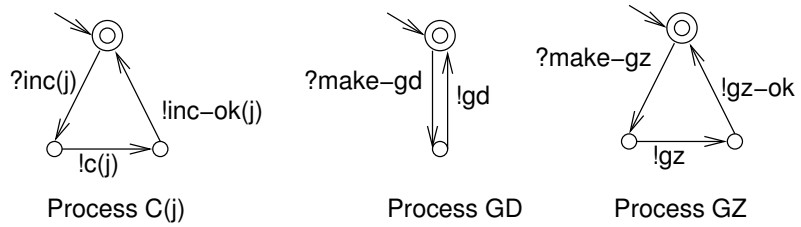

Fig. 9. Description of processes $GD, GZ, C(j)$ in Phase (ii)

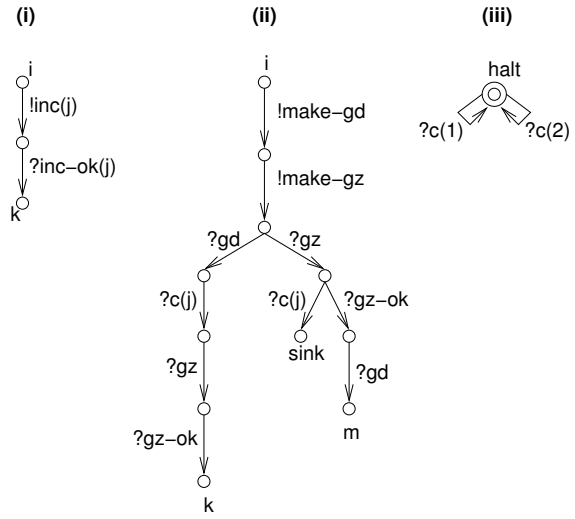

Fig. 10. Transitions of process A in Phase (ii)

outgoing transitions. As in Phase (i), for the halting state of $\mathcal{M}$, A contains two special transitions, shown in Fig. 10(iii). It is clear that the transitions of $\mathcal{A}$ now strictly adhere to the definition of systems in Section 2.

Let Sch be a valid schedule for $\mathcal{A}$. As in Phase (i), we show that Sch guides $\mathcal{A}$ to simulate correctly the execution of $\mathcal{M}$. The simulation of an $(i, Inc(j), k)$ instruction is as in Phase (i). Now suppose that, according to Sch, the execution of the system $\mathcal{A}$ arrives at a configuration in which A is at state i, the corresponding instruction of $\mathcal{M}$ is $(i, Dec(j), k, m)$, and each of processes GD, GZ is at its initial state. Then it is not difficult to see that Sch must first select process A twice in a row to execute $!make\text{-}gd$, $!make\text{-}gz$ transitions, and thus GD, GZ become enabled. Next, suppose $c(j)$ is greater than zero. Then as in Phase (i), Sch has to guide A to execute *only* the transition $?gd$, and eventually, $c(j)$ is decremented, A moves to state k, and GD, GZ return to their initial states. Now suppose that $c(j)$ is zero, then Sch has to guide A to execute *only* the transition $?gz$, and eventually, $c(j)$ remains zero, A moves to state m, and GD, GZ return to their initial states.

With the observation that any valid schedule Sch guides $\mathcal{A}$ to simulate the execution of $\mathcal{M}$, it follows from similar arguments as in Phase (i) that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable. —**End of Phase (ii)**

—**Phase (iii)**: Finally, we modify the construction of the system $\mathcal{A}$ defined in Phase (ii) so that it becomes deadend-free and still, any valid and regular schedule for $\mathcal{A}$ simulates the execution of $\mathcal{M}$. One can then show that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable. We first explain the final construction of $\mathcal{A}$, then argue that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable, and last show that $\mathcal{A}$ is deadend-free.

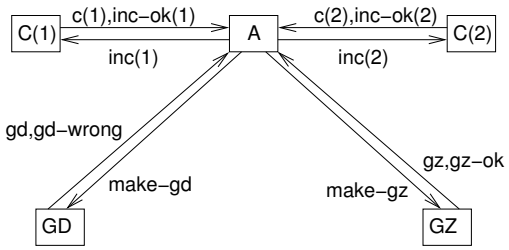

Fig. 11. The architecture of $\mathcal{A}$ in Phase (iii)

The communication architecture of $\mathcal{A}$ is now as shown in Fig. 11. The transitions of GD, GZ and processes $C(j)$, $j = 1, 2$, are displayed in Fig. 12. The transitions of A are depicted in Fig. 13. For each $(i, Inc(j), k)$ instruction of $\mathcal{M}$, A contains the transitions shown in Fig. 13(i). For each $(i, Dec(j), k, m)$ instruction of $\mathcal{M}$, A contains the transitions shown in Fig. 13(ii), where *sink* is a distinguished state, present also in Fig. 13(iii). For the states *sink* and *halt*, A contains the special transitions shown in Fig. 13(iii) (where unlabelled arrows bear implicit labels in Σ^{cho}).

We first note that the special transitions in Fig. 13(iii) are designed in such a way that any valid and regular schedule *cannot* lead $\mathcal{A}$ to a configuration in which process A is at the state *sink*. To see this, suppose Sch is a valid and regular schedule for $\mathcal{A}$. Assume further that according to Sch, $\mathcal{A}$ arrives at a configuration in which process A is at the state *sink*. Recall that Sch can *not* discriminate between the two outgoing transitions of *sink* which are (data-dependent) choice transitions. Thus, Sch has to allow runs in which the transitions $!inc(1)$, $?inc\text{-}ok(1)$ of Fig. 13(iii) are executed arbitarily many times, in tight interleaving with corresponding transitions $?inc(1)$, $!c(1)$, $!inc\text{-}ok(1)$ from $C(1)$. Thus, Sch allows complete runs in which A is arbitrarily often at state *sink* and the size of $c(1)$ can be arbitarily large. Consequently, Sch is not regular, a contradiction.

By the above observation that any valid and regular schedule for $\mathcal{A}$ drives $\mathcal{A}$ so as to avoid visiting *sink*, similar arguments as in Phase (ii) may be used to show that any valid and regular schedule for $\mathcal{A}$ guides $\mathcal{A}$ to simulate correctly the execution of $\mathcal{M}$. Now, as in Phase (i), if $\mathcal{M}$ halts, then one can construct

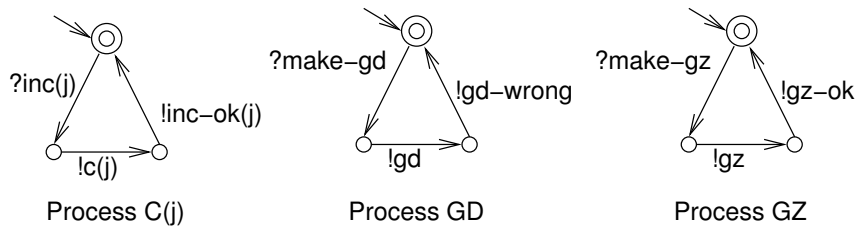

Fig. 12. Description of processes $GD, GZ, C(j)$ in Phase (iii)

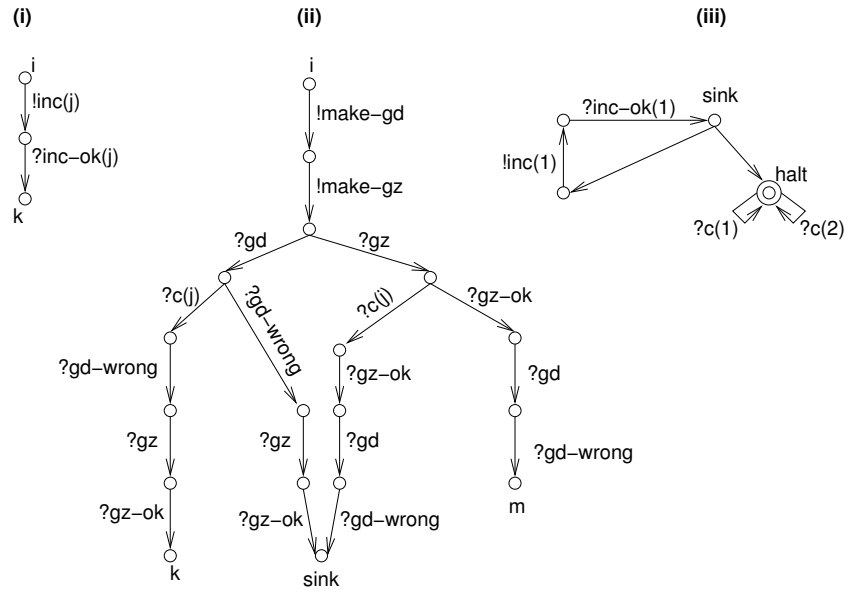

Fig. 13. Transitions of process A in Phase (iii)

a valid and regular schedule which leads $\mathcal{A}$ to the configuration in which each process is at its final state, thus in particular A is in state *halt*, and all buffers except possibly $c(1), c(2)$ are empty. Further, during the execution of $\mathcal{A}$ under Sch, A never visits state *sink*. With the special transitions shown in Fig. 13(iii), A will eventually also empty buffers $c(1), c(2)$ under the schedule Sch. Thus $\mathcal{A}$ is schedulable. On the other hand, if $\mathcal{M}$ does not halt, then under any valid schedule of $\mathcal{A}$, process A must nevertheless reach the *halt* state and hence visit the *sink* state, and therefore $\mathcal{A}$ does not have any valid and *regular* schedule. We have thus shown that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable.

Finally, we show that the system $\mathcal{A}$ constructed in this final phase is deadend-free. We assume that from any control state i of $\mathcal{M}$ except the halting state, it is always possible to reach a control state t with a corresponding instruction of the form $(t, Dec(j), k, m)$. This assumption can be made without any loss of generality, since one may always replace each $(i, Inc(j), k)$ instruction by an equivalent sequence of three instructions $(i, Inc(j), i')$, $(i', Inc(j), i'')$, $(i'', Dec(j), k, k)$ where i',i'' are new control states with $i' \neq i''$.

To prove that $\mathcal{A}$ is deadend-free, we need to show that every run σ of the *unscheduled* system $\mathcal{A}$ can be extended to a complete run. We proceed by cases according to whether $\mathcal{M}$ halts or not, and in each case we consider two types of runs of $\mathcal{A}$ (notice that runs which are neither of type I or II are not schedulable):

Type I: Runs which simulate the execution of $\mathcal{M}$ so that A never visits state *sink*.

Type II: Runs which end at the configuration in which A is at state *sink*,

every other process is at its initial state, and all buffers except possibly $c(1), c(2)$ are empty.

—**Case (i):** $\mathcal{M}$ halts.

Let σ be a run of $\mathcal{A}$. If σ is of type I, then in this run, process A reaches the state *halt* without going through the state *sink*, and by scheduling process A until $c(1)$ and $c(2)$ are empty, σ can be extended to a complete run of $\mathcal{A}$. If σ is of type II, then σ can be extended to a run ending at the configuration in which A is at state *halt*, every other process is at its initial state, and all buffers except possibly $c(1), c(2)$ are empty. The run σ can therefore be extended to a complete run in which process A is scheduled finally until these two buffers are also empty.

—**Case (ii)**: $\mathcal{M}$ does not halt.

Let σ be a run of $\mathcal{A}$. First consider the case where σ is of type I. As said above, we can assume that from any control state i of $\mathcal{M}$ except the halting state, it is always possible to reach a control state î whose corresponding instruction has the form $(\hat{\imath}, Dec(j), k, m)$. Thus, σ can certainly be extended to a run σ' that ends at a configuration in which A is at some state i and the corresponding instruction of $\mathcal{M}$ is of the form $(i, Dec(j), k, m)$. In view of Fig. 13(ii), σ' can be extended further to a run σ'' that ends at a configuration in which A is at state *sink*. In view of Fig. 13(iii), σ'' can be extended in turn to a complete run.

For the case where σ is of type II, the same arguments as in case (i) show that σ can be extended to a complete run.

—**End of Phase (iii)**

With the construction of a deadend-free system $\mathcal{A}$ such that $\mathcal{M}$ halts iff $\mathcal{A}$ is schedulable, we have completed the proof of Thm. 1.

4 Data-Branching and Decidability.

The ability of a schedule to bias the choice between two receive actions (e.g. $?gd$ and $?gz$) of the same process is crucial to our undecidability proof. This observation leads us to consider a restricted class of systems as follows. A system $\mathcal{A}$ is *data-branching* if for each process p and for each state $s_p \in S_p$, if s_p is a polling state, then it has exactly one outgoing transition. Thus the only branching states are those at which internal (data-dependent) choices take

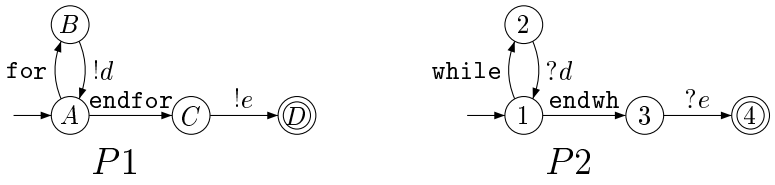

Fig. 14. A data-branching system.

place. For instance, the task system shown in Fig. 3 is not data branching, since process $P2$ chooses between receiving from channels d or e in state 1. However, an implementation of this system depicted in Fig. 14 is data-branching. More generaly, Kahn process networks that have a terminating semantics and in which numerical data has been abstracted away, are data-branching. This implies for such a class of restricted Kahn networks, that QSS is decidable. A similar property does not hold for the YAPI extension of Kahn networks considered in [4].

The question arises whether the quasi-static scheduling problem for data-branching systems is decidable. We show that the answer is yes. This result subsumes the similar result obtained in [12] for systems without loops.

Theorem 2 *Given a data-branching system $\mathcal{A}$, one can effectively decide whether $\mathcal{A}$ is schedulable.*

The rest of this section is devoted to the proof of Thm. 2. We assume throughout that $\mathcal{A}$ is data-branching. The proof relies crucially on the notion of a *canonical schedule* for $\mathcal{A}$, denoted Sch_{ca}. The canonical schedule is *positional* (also called memoryless in the literatute), that is, $Sch_{ca}(\sigma) = Sch_{ca}(\sigma')$ whenever runs σ,σ' end at the same configuration. Thus, we consider Sch_{ca} as a function from RC to $\mathcal{P}$. Informally, at configuration (s, χ), if there is a $p \in \mathcal{P}$ such that p is enabled and s_p is a polling or choice state, then Sch_{ca} picks one such p. If there is no such process, then for each process p enabled at (s, χ), s_p has exactly one outgoing transition $(s_p, !c_p, s'_p)$. In this case, Sch_{ca} picks a process p with $\chi(c_p)$ being minimal. Ties will be broken by fixing a linear ordering on $\mathcal{P}$. The proof of Thm. 2 consists of two steps. Firstly, we show that $\mathcal{A}$ is schedulable iff Sch_{ca} is a valid and regular schedule (Prop. 1). Secondly, we prove that one can effectively decide whether Sch_{ca} is a valid and regular schedule (Thm. 4).

4.1 The Canonical Schedule.

We fix a total order $\leq_{\mathcal{P}}$ on $\mathcal{P}$ and define the *canonical schedule* Sch_{ca} for $\mathcal{A}$ as follows. For each configuration (s, χ), let $P^{(s,\chi)}_{enable} \subseteq \mathcal{P}$ be the set of processes

enabled at (s, χ). We partition $P^{(s,\chi)}_{enable}$ into $P^{(s,\chi)}_{poll}$, $P^{(s,\chi)}_{choice}$ and $P^{(s,\chi)}_{send}$ as follows. For $p \in P^{(s,\chi)}_{enable}$, we have:(i) $p \in P^{(s,\chi)}_{poll}$ iff s_p is a polling state; (ii) $p \in P^{(s,\chi)}_{choice}$ iff s_p is a choice state; (iii) $p \in P^{(s,\chi)}_{send}$ iff s_p is a sending state. We further define the set $P^{(s,\chi)}_{send\text{-}min} \subseteq P^{(s,\chi)}_{send}$ as follows: for $p \in P^{(s,\chi)}_{send}$, we have $p \in P^{(s,\chi)}_{send\text{-}min}$ iff $\chi(c_p) \leq \chi(c_q)$ for each $q \in P^{(s,\chi)}_{send}$, where $!c_p$ (respectively, $!c_q$) is the action of p (respectively, of q) enabled at (s, χ).

The canonical schedule Sch_{ca} maps each configuration (s, χ) to the process $Sch_{ca}(s, \chi)$ as follows. If $P^{(s,\chi)}_{poll} \cup P^{(s,\chi)}_{choice} \neq \emptyset$, then $Sch_{ca}(s, \chi)$ is the least member of $P^{(s,\chi)}_{poll} \cup P^{(s,\chi)}_{choice}$ with respect to $\leq_{\mathcal{P}}$. Otherwise, $Sch_{ca}(s, \chi)$ is the least member of $P^{(s,\chi)}_{send\text{-}min}$ with respect to $\leq_{\mathcal{P}}$. It is straightforward to verify that Sch_{ca} adheres to the definition of schedules. We say that a schedule Sch' is *optimal* if for every run σ in Run/Sch', every channel c, and every schedule Sch, there exists a run $\tau \in Run/Sch$ and a channel d with $\chi^{\sigma}(c) \leq \chi^{\tau}(d)$.

Proposition 1 *A data-branching system $\mathcal{A}$ is schedulable iff Sch_{ca} is a valid and regular schedule for $\mathcal{A}$. Furthermore, Sch_{ca} is optimal.*

To facilitate the proof of Prop. 1, we introduce now an equivalence on complete runs. For $\sigma \in \Sigma^{\star}$ and $p \in \mathcal{P}$, let $prj_p(\sigma)$ be the sequence obtained from σ by erasing letters not in Σ_p. We define the equivalence relation $\sim \subseteq Run_{cpl} \times Run_{cpl}$ as follows: $\sigma \sim \sigma'$ iff for every $p \in \mathcal{P}$, $prj_p(\sigma) = prj_p(\sigma')$. We observe a useful relation between $\sim$ and schedules.

Lemma 1 *Let σ be a complete run of a data-branching system $\mathcal{A}$. Suppose that Sch is a schedule of $\mathcal{A}$ (not necessarily valid nor regular). Then there exists a complete run σ' such that $\sigma' \sim \sigma$ and $\sigma' \in Run/Sch$.*

Proof. Let $\sigma = \tau a \tau'$, with $a \in \Sigma_p$, $\tau \in Run/Sch$, and $Sch(\tau) = q \neq p$. In particular, $\tau a \notin Run/Sch$ and q is enabled at (s^{τ}, χ^{τ}). We show that there exists a complete run w of the form $\tau b \tau''$ with $b \in \Sigma_q$ (thus τb is in agreement with Sch) and $w \sim \sigma$. Repeating inductively this argument yields eventually the desired complete run σ' in agreement with Sch such that $\sigma' \sim \sigma$.

Note that, by the completeness of the run σ, s^{σ}_q is the final state of q. It thus follows from the definition of a task system that, either s^{σ}_q has no outgoing transitions, or s^{σ}_q is a polling state. In order to show the existence of w as above, we consider two cases.

—**Case** (i): s^{τ}_q is a sending state or a choice state.

We have $s^{\tau}_q \neq s^{\sigma}_q$ since s^{σ}_q either has no outgoing transitions or it is a polling state. So some (choice or sending) action b in Σ_q should occur in τ' to move process q from s^{τ}_q. Let $\tau' = \rho b \rho'$ where ρ contains no letter of Σ_q. Then one

readily verifies that $w = \tau b a \rho \rho'$ is also a run of $\mathcal{A}$ and that $w \sim \sigma$.

—**Case** (ii): s_q^τ is a polling state.

Since $Sch(\tau) = q$, some action $?c$ with $dst(c) = q$ is enabled at the configuration (s^τ, χ^τ). That is, $(s_q^\tau, ?c, s_q)$ is an outgoing transition of s_q^τ and $\chi^\tau(c) > 0$. We show that $?c$ occurs in τ' and thus if we write τ' in the form of $\rho\, ?c\, \rho'$ where ρ contains no letter of Σ_q, then $w = \tau\, ?c\, a \rho \rho'$ is also a run of $\mathcal{A}$ and $w \sim \sigma$. Since $\mathcal{A}$ is data-branching and s_q^τ is a polling state, if some action in τ' belongs to Σ_q, then the first such action must be $?c$. Towards a contradiction, suppose that there is no action of q in τ', then $s_q^\tau = s_q^\sigma$ and there is no $?c$ in τ' (since $?c$ is an action of q), hence $\chi^\sigma(c) \geq \chi^\tau(c) > 0$ contradicting the fact that (s^σ, χ^σ) is a final configuration. □

Observation 3 *Lemma 1 implies that a run σ in Run/Sch can be extended to a run in Run_{cpl}/Sch iff it can be extended to a run in Run_{cpl}. This holds for every schedule Sch (not necessarily valid nor regular), provided that the system is data-branching.*

The specific power of the valid schedules is shown by the lemma below.

Lemma 2 *If there exists a valid schedule Sch, then the (unscheduled) data-branching system is deadend free, i.e. any run may be extended to a complete run.*

Proof. Let σ be a run. Consider the following algorithm:

Let $\rho := \varepsilon$
For each $p \in \mathcal{P}$, $w_p := prj_p(\sigma)$
while $\rho \notin Run_{cpl}$ **do**
  if $Sch(\rho) = p$ **then**
    if $w_p = a_p w'_p$ **then**
      begin $\rho := \rho a_p$; $w_p := w'_p$ **end**
    else $\rho := \rho a_p$ **for some** $a_p \in \Sigma_p$ **such that** $\rho a_p \in Run$
done

As Sch is valid, any run in Run/Sch may be extended to a run in $Run_{cpl} \cap (Run/Sch)$, and this algorithm has at least one terminating execution, leading to a complete run ρ. Let σ' be the largest prefix of σ such that for each $p \in \mathcal{P}$, $prj_p(\sigma')$ is a prefix of $prj_p(\rho)$. We prove now that $\sigma' = \sigma$. It implies that for all p, $prj_p(\sigma)$ is a prefix of $prj_p(\rho)$, and thus that σ can be extended to some complete run $\rho' \sim \rho$.

Assume by contradiction that $\sigma' \neq \sigma$, and let $\sigma = \sigma' a \sigma''$. Let $q \in \mathcal{P}$ with $a \in \Sigma_q$. By definition of the algorithm, $prj_q(\sigma') = prj_q(\rho)$. As ρ is complete,

the state $s_q^{\sigma'}$ reached on process q after doing σ' is a final state, and in particular it is polling. As $\sigma' a$ is a run, $a = ?c$ for some channel c with $dst(c) = q$. Let $src(c) = p$. For each letter $b \in \Sigma$ and run ω, let $\#_b(\omega)$ denote the number of occurrences of letter b in sequence ω. As $prj_p(\sigma')$ is a prefix of $prj_p(\rho)$, we have $\#_{!c}(\sigma') \leq \#_{!c}(\rho)$. As ρ is complete, $\#_{!c}(\rho) = \#_{?c}(\rho)$. As $prj_q(\sigma') = prj_q(\rho)$, we have $\#_{?c}(\rho) = \#_{?c}(\sigma')$. Combining them, we get $\#_{!c}(\sigma') \leq \#_{?c}(\sigma')$. It means that $\#_{!c}(\sigma'\ a) < \#_{?c}(\sigma'\ a)$ as $a = ?c$. Since $\sigma' a$ is a run, $\#_{!c}(\sigma'\ a) \geq \#_{?c}(\sigma'\ a)$, a contradiction. □

Using Lemmas 1 and 2, it is easy to show that if there exists a valid schedule Sch, then the canonical schedule Sch_{ca} is valid too.

Lemma 3 *A data-branching system $\mathcal{A}$ admits some valid schedule iff Sch_{ca} is valid for $\mathcal{A}$.*

Proof. It suffices to consider the "only if" direction. Let Sch be a valid schedule for $\mathcal{A}$, and let σ be any run in Run/Sch_{ca}. By Lemma 2, there exists a continuation τ such that $\sigma\tau \in Run_{cpl}$. By Observation 3, σ may be extended to a run in $Run_{cpl} \cap (Run/Sch_{ca})$, hence the canonical schedule is valid. □

The concept of an *anchored* run, that we introduce now will also play a crucial role in what follows. If χ is a mapping from Ch to the non-negative integers, let $\max(\chi) = \max\{\chi(c) \mid c \in Ch\}$. For a run σ, we define the *height* $\max(\sigma)$ of the run σ by $\max(\sigma) = \max\{\max(\chi^{\sigma'}) \mid \sigma' \text{ is a prefix of } \sigma\}$. We say that σ is an *anchored* run iff σ *is non-empty* and denoting $\sigma = \sigma' a$ with $a \in \Sigma$, $\max(\sigma) > \max(\sigma')$. That is, a run is anchored if the height of the run has just been strictly increased. Anchored runs in agreement with Sch_{ca} have a special property: every action enabled concurrently with the last action of an anchored run is a send action on some buffer that holds a maximum number of items. This property may be stated precisely as follows.

Lemma 4 *Let σ be an anchored run according to Sch_{ca}, and let $M = \max(\sigma)$. Then $\sigma = \hat{\sigma}!c$ for some $c \in Ch$ and $\chi^{\sigma}(c) = M$. Further, if $a \in \Sigma$ is enabled at $(s^{\hat{\sigma}}, \chi^{\hat{\sigma}})$, then $a = !d$ for some $d \in Ch$ and moreover $\chi^{\hat{\sigma}}(d) = M - 1$. In particular, $\chi^{\hat{\sigma}}(c) = M - 1$.*

We are now ready to prove Prop. 1.

Proof. of Prop. 1

The if part is obvious. As for the only if part, let Sch be a valid and regular schedule for $\mathcal{A}$. First, it follows from Lemma 3 that Sch_{ca} is valid.

We prove that Sch_{ca} is regular. We know that RC/Sch contains a finite number k of configurations. Since each action adds at most one item to one buffer, for all $\sigma \in Run/Sch$, $\max(\sigma) \leq k$. We will prove that for all $\sigma_{ca} \in Run/Sch_{ca}$,

$\max(\sigma_{ca}) \leq \max(\sigma) \leq k$, which will imply that RC/Sch_{ca} has a finite number of configurations. It also implies that Sch_{ca} is optimal. Since we know that Sch_{ca} is valid, it suffices to consider only complete runs in Run/Sch_{ca}.

Let $\sigma_{ca} \in Run/Sch_{ca}$ be a complete run. Relying on Lemma 1, let $\sigma \in Run/Sch$ be a complete run such that $\sigma \sim \sigma_{ca}$. Suppose $M_{ca} = \max(\sigma_{ca})$ and $M = \max(\sigma)$. Pick the least prefix τ_{ca} of σ_{ca} such that $\tau_{ca} = M_{ca}$. Thus τ_{ca} is anchored. By Lemma 4, let $\tau_{ca} = \hat{\tau}_{ca}!c$. Consider the sequence $\hat{\tau}_{ca}$. For a run $\tau \in Run$, we say τ is *covered* by $\hat{\tau}_{ca}$ iff for every $p \in \mathcal{P}$, the projection $prj_p(\tau)$ of τ on $(\Sigma_p)^*$ is a prefix of $prj_p(\hat{\tau}_{ca})$. Now pick τ as the least prefix of σ such that τ is *not* covered by $\hat{\tau}_{ca}$. Such a τ exists, following the definition of $\sim$. Let $\tau = \hat{\tau}a$ where $a \in \Sigma$ is the last letter of τ. Let $p_a = p$ such that $a \in \Sigma_p$. We consider three cases.

—**Case (i)** $a = !d$ for some $d \in Ch$.

The choice of τ implies $prj_{p_a}(\hat{\tau}) = prj_{p_a}(\hat{\tau}_{ca})$. Thus, $s^{\hat{\tau}}(p_a) = s^{\hat{\tau}_{ca}}(p_a)$. And $!d$ is enabled at configuration $(s^{\hat{\tau}_{ca}}, \chi^{\hat{\tau}_{ca}})$. It follows from Lemma 4 that $\chi^{\hat{\tau}_{ca}}(d) = M_{ca} - 1$ (whether $d = c$ or not). As $dst(d) \neq p_a$, the choice of τ also implies $prj_{dst(d)}(\hat{\tau})$ is a prefix of $prj_{dst(d)}(\hat{\tau}_{ca})$. Hence, we have $\#_{!d}(\hat{\tau}) = \#_{!d}(\hat{\tau}_{ca})$ and $\#_{?d}(\hat{\tau}) \leq \#_{?d}(\hat{\tau}_{ca})$. It follows that $\chi^{\hat{\tau}}(d) \geq \chi^{\hat{\tau}_{ca}}(d)$. Combining these observations with $\chi^{\hat{\tau}}(d) \leq M - 1$ then yields $M_{ca} \leq M$.

–**Case (ii)**: $a = ?d$ for some $d \in Ch$.

By the same argument as in case (i), we have $s^{\hat{\tau}}(p_a) = s^{\hat{\tau}_{ca}}(p_a)$. Also we have $prj_{p_a}(\hat{\tau}) = prj_{p_a}(\hat{\tau}_{ca})$, and $prj_{src(d)}(\hat{\tau})$ is a prefix of $prj_{src(d)}(\hat{\tau}_{ca})$. Hence, $\chi^{\hat{\tau}}(d) \leq \chi^{\hat{\tau}_{ca}}$. It follows that $?d$ is enabled at configuration $(s^{\hat{\tau}_{ca}}, \chi^{\hat{\tau}_{ca}})$. This contradicts that at configuration $(s^{\hat{\tau}_{ca}}, \chi^{\hat{\tau}_{ca}})$, the schedule Sch_{ca} picks process $src(c)$ with $s^{\hat{\tau}_{ca}}(src(c))$ being a sending state.

—**Case (iii)**: $a \in \Sigma^{cho}_{p_a}$.

Similar to Case (ii), we obtain a contradiction by noting that a is enabled at $(s^{\hat{\tau}_{ca}}, \chi^{\hat{\tau}_{ca}})$. □

4.2 Deciding Boundedness of the Canonical Schedule.

The decision procedure for the boundedness of Sch_{ca} is similar to the decision procedure for the boundedness of Petri nets [10]. We now briefly recall the outline of the classical algorithm defined in [10]. First, an order $\sqsubseteq$ on runs is defined, such that $\sigma \sqsubseteq \sigma'$ if the following conditions hold:

- σ is a strict prefix of σ'.
- $s^{\hat{\sigma}}(p) = s^{\hat{\sigma}'}(p)$ for every $p \in \mathcal{P}$.
- $\chi^{\sigma}(d) \leq \chi^{\sigma'}(d)$ for each $d \in Ch$.

For two runs σ, σ', define $\sigma \equiv \sigma'$ if $(s^{\sigma}, \chi^{\sigma}) = (s^{\sigma'}, \chi^{\sigma'})$, that is if both runs end at the same configuration. First, [10] shows that $\sqsubseteq \cup \equiv$ is a well quasi order, which implies by König's lemma that the tree of runs built inductively by extending every run σ' by one step unless $(\sigma, \sigma') \in (\sqsubseteq \cup \equiv)$ for some run σ already present, is finite. Second, [10] shows that $\sigma \sqsubseteq \sigma'$ witnesses for the unboundedness of the system. Intuitively, when $\sigma \sqsubseteq \sigma'$ and $\sigma' = \sigma\tau$, one can iterate τ to increase at least one buffer beyond any bound.

Notice that one cannot apply directly the classical algorithm defined in [10] to check the boundedness of the canonical schedule, because RC/Sch_{ca} cannot be represented as the set of reachable markings of a Petri net. Indeed, the canonical schedule performs a zero-test when it schedules a process ready to send, because it must check that all processes ready to receive have empty input buffers. We show that one can nevertheless define a quasi order $\prec_{ca}$ on runs, and build a *finite* tree in the same way as in [10] (Proposition 2), such that $\sigma \prec_{ca} \sigma'$ for two runs σ, σ' in this tree iff RC/Sch_{ca} is not a finite set or Sch_{ca} is not a valid schedule for $\mathcal{A}$ (Proposition 3).

More precisely, let the quasi order $\prec_{ca}$ be defined such that $\sigma \prec_{ca} \sigma'$ iff $\sigma \sqsubseteq \sigma'$, both σ, σ' are *anchored* (in particular they are non empty), and both runs end with the same action, that is $\sigma = \hat{\sigma}!c$, $\sigma' = \hat{\sigma}'!c$ for some $c \in Ch$. The general intuition is as follows. If $\sigma \prec_{ca} \sigma'$ and $\sigma' = \sigma\tau$, then for any n, even though the anchored run $\sigma(\tau)^n$ is not necessarily in agreement with the canonical schedule, there exists a continuation w_n such that $\sigma(\tau)^n w_n \in Run_{cpl}$ and $\sigma(\tau)^n w_n \sim \rho_n$ for some run $\rho_n \in Run/Sch_{ca}$ with height $max(\rho_n)$ at least equal to $max(\sigma(\tau)^n)$, and hence larger than n.

Notice that for $\sigma \prec_{ca} \sigma'$, in particular, $\chi^{\sigma}(c) < \chi^{\sigma'}(c)$ since σ is a strict prefix of σ' and both are anchored. We show now a structural property of $\prec_{ca}$ which will serve us to produce a *finite* coverability tree of all runs. An *infinite* run of $\mathcal{A}$ is an infinite sequence ρ in Σ^{ω} such that every finite prefix of ρ is in $Run(\mathcal{A})$. We say that an infinite run ρ agrees with Sch_{ca} iff every finite prefix of ρ agrees with Sch_{ca}.

Proposition 2 *Let $\rho \in \Sigma^{\omega}$ be an infinite run in agreement with Sch_{ca}. Then there exist two finite prefixes σ,σ' of ρ such that either σ, σ' end at the same configuration, or $\sigma \prec_{ca} \sigma'$ (in which case σ, σ' are both anchored).*

Proof. If there exists $k \in \mathbb{N}$ such that for all prefixes α of ρ, $max(\chi^{\alpha}) \leq k$, then there is only a finite number of possible configurations reached during ρ, hence we can find two prefixes of ρ ending at the same configuration. Otherwise, $max(\chi^{\alpha})$ is unbounded, and one can extract from ρ an infinite subsequence

of *anchored* prefixes. Since there is a finite number of buffers and a finite number of tuples of local states in $\Pi_{p\in\mathcal{P}}(S_p)$, one can extract from ρ an infinite subsequence of anchored prefixes with the same maximal channel $c \in Ch$ and the same tuple of local states $s \in \Pi_{p\in\mathcal{P}}(S_p)$.

By an inductive argument on $i \leq |Ch|$, one easily verifies that there exists an infinite subsequence of anchored prefixes $\alpha_0, \alpha_1, \cdots$ of ρ, such that $\chi^{\alpha_0}(c_j) \leq \chi^{\alpha_1}(c_j) \leq \ldots$ for every index $1 \leq j \leq i$. In particular, this shows the existence of prefixes σ, σ' of ρ such that $\sigma \prec_{ca} \sigma'$. □

Next we show that any pair of runs σ, σ' in Run/Sch_{ca} such that $\sigma \prec_{ca} \sigma'$ witnesses for the unboundedness of RC/Sch_{ca} (or for the non-validity of Sch_{ca}). This requires a new argument not in [10] because, even though $\sigma' = \sigma\tau$ and both σ, σ' agree with Sch_{ca}, the run $\sigma\tau^n$ may disagree with Sch_{ca} for some n. However, we shall argue that if there exist two anchored runs satisfying $\sigma \prec_{ca} \sigma'$ then for every $n = 1, 2, \ldots$, there exists a run ρ_n according to Sch_{ca} such that either $\max(\rho_n) \geq n$ or ρ_n cannot be extended to reach a final configuration, entailing by Lemma 2 that Sch_{ca} is not valid.

Proposition 3 *If there exist two anchored runs σ, σ' in Run_{an}/Sch_{ca} such that $\sigma \prec_{ca} \sigma'$, then either RC/Sch_{ca} has an infinite number of configurations or Sch_{ca} is not valid.*

Proof. Let $\sigma' = \sigma\tau$. Fix an arbitrary integer $k > 1$ and consider the sequence $\alpha = \sigma\tau\tau\ldots\tau$ (k copies of τ). Following the definition of $\prec_{ca}$, one verifies that α is a run of $\mathcal{A}$. If α cannot be extended to a complete run, then by Lemma 2, Sch_{ca} is not valid and this concludes the proof. Otherwise, by Lemma 1, there exists a continuation $w \in \Sigma^*$ such that αw is a complete run and $\alpha w \sim \rho$ for some run $\rho \in Run_{cpl} \cap (Run/Sch_{ca})$. Let $M = \max(\sigma)$ and $M' = \max(\sigma')$. Let $\sigma = \hat{\sigma}!c$, $\sigma' = \hat{\sigma}'!c$, where $c \in Ch$, $\chi^{\sigma}(c) = M$, $\chi^{\sigma'}(c) = M'$. We show below that $\max(\rho) \geq M + k \cdot (M' - M)$ and thus Sch_{ca} is not regular.

Though $\sigma\tau$ agrees with Sch_{ca}, we note that α is not necessarily a prefix of ρ. Let $\alpha = \hat{\alpha}!c$. Consider the sequence $\hat{\alpha}$. For a prefix β of ρ, recall from the proof of Prop. 1 that β is covered by $\hat{\alpha}$ iff for every $p \in \mathcal{P}$, $prj_p(\beta)$ is a prefix of $prj_p(\hat{\alpha})$. Pick β to be the least prefix of ρ such that β is *not* covered by $\hat{\alpha}$. Let $\beta = \hat{\beta}b$ where b is the last letter of β. Let $p_b \in \mathcal{P}$ be the process such that $b \in \Sigma_{p_b}$. The choice of β implies that $prj_{p_b}(\hat{\beta}) = prj_{p_b}(\hat{\alpha})$, and thus $s^{\hat{\beta}}(p_b) = s^{\hat{\alpha}}(p_b)$. Again we consider three cases.

—**Case** (i). $b = !d$ for some $d \in Ch$.

Thus, $!d$ is enabled at configuration $(s^{\hat{\alpha}}, \chi^{\hat{\alpha}})$. Also, as $dst(d) \neq p_b$, we have that $prj_{dst(d)}(\hat{\beta})$ is a prefix of $prj_{dst(d)}(\hat{\alpha})$. Thus, we have $\#_{!d}(\hat{\beta}) = \#_{!d}(\hat{\alpha})$, and $\#_{?d}(\hat{\beta}) \leq \#_{?d}(\hat{\alpha})$, where $\#_a(\theta)$ denotes the number of occurrences of letter a

in sequence θ. It follows that $\chi^{\hat{\beta}}(d) \geq \chi^{\hat{\alpha}}(d)$.

Note that $\chi^{\hat{\alpha}}(c) = M + k \cdot (M' - M) - 1$ and $\chi^{\hat{\beta}}(d) \leq \max(\rho) - 1$. Thus, if $d = c$, then we have $\max(\rho) \geq M + k \cdot (M' - M)$. Otherwise, $d \neq c$. By definition of β, $!d$ is enabled at $(s^{\hat{\alpha}}, \chi^{\hat{\alpha}})$, hence seeing that $s^{\hat{\alpha}} = s^{\hat{\sigma}'}$, $!d$ is enabled also at $(s^{\hat{\sigma}'}, \chi^{\hat{\sigma}'})$. By definition of $\prec_{ca}$, we conclude that $!d$ is also enabled at $(s^{\hat{\sigma}}, \chi^{\hat{\sigma}})$. Thus, $\chi^{\hat{\sigma}}(d) = M - 1$ and $\chi^{\hat{\sigma}'}(d) = M' - 1$, owing to the fact that $\hat{\sigma}!c$ and $\hat{\sigma}'!c$ agree with Sch_{ca} and in view of Lemma 4. It follows that $\chi^{\hat{\alpha}}(d) = M - 1 + k \cdot (M' - M)$. Consequently, we also have $\max(\rho) = M + k \cdot (M' - M)$.

—**Case** (ii). $b = ?d$ for some $d \in Ch$.

From the definition of $\prec_{ca}$, $s^{\hat{\sigma}}(p_b) = s^{\hat{\sigma}'}(p_b) = s^{\hat{\alpha}}(p_b) = s^{\hat{\beta}}(p_b)$. At configuration $(s^{\hat{\sigma}}, \chi^{\hat{\sigma}})$, Sch_{ca} picks process $src(c)$ where $s^{\hat{\sigma}}(src(c))$ is a sending state. Hence, p_b is not enabled at $(s^{\hat{\sigma}}, \chi^{\hat{\sigma}})$. That is, $\chi^{\hat{\sigma}}(d) = 0$. Similarly, $\chi^{\hat{\sigma}'}(d) = 0$. As a result, $\chi^{\hat{\alpha}}(d) = 0$.

However, by similar arguments as in case (i), one sees that $\#_{?d}(\hat{\beta}) = \#_{?d}(\hat{\alpha})$ and $\#_{!d}(\hat{\beta}) \leq \#_{!d}(\hat{\alpha})$. Thus, $\chi^{\hat{\beta}}(d) \leq \chi^{\hat{\alpha}}(d)$, and $\chi^{\hat{\beta}}(d) = 0$, in contradiction with the fact that β is a run.

—**Case** (iii). $b \in \Sigma^{cho}_{p_b}$.

As in Case (ii), we derive a contradiction by noting that p_b is enabled at $(s^{\hat{\sigma}}, \chi^{\hat{\sigma}})$, becauses $s^{\hat{\alpha}}(p_b) = s^{\hat{\beta}}(p_b)$. □

The set of all runs of a data-branching system under the canonical schedule Sch_{ca} forms a possibly infinite tree (in which any data dependent choice performed by a scheduled process induces several branches). Following Karp and Miller's ideas, one may stop exploring this tree whenever coming again to a configuration already visited within some run in the tree, or reaching an anchored run σ' that extends a smaller anchored run σ, i.e. $\sigma \prec_{ca} \sigma'$. Based on this construction of a finite coverability tree, we obtain the following theorem.

Theorem 4 *One can effectively determine whether Sch_{ca} is valid and regular.*

Proof. We construct inductively as follows a tree W of valid runs in agreement with Sch_{ca}. First, ε is in W. Then, suppose that σ is in W and σa is a run in agreement with Sch_{ca}, with $a \in \Sigma$. If there exists $\sigma' \in W$ such that $\sigma' \prec_{ca} \sigma a$, then we can stop the construction of W and report that either Sch_{ca} is not regular or Sch_{ca} is not valid, based on Prop. 3. Otherwise, we check whether there exists $\tau \in W$ such that τ ends at the same configuration as σa. If such a τ does not exist, then we add σa to W (otherwise we just ignore σa).

We first prove that the construction of W stops after a finite number of steps. Suppose otherwise. Then the runs in W form an infinite tree. By König's lemma, there exists an infinite sequence ρ of Σ^ω such that every finite prefix of ρ is in W. Applying Prop. 2, we get that there exist two finite prefixes σ,σ' of ρ such that σ is a prefix of σ' and either σ, σ' end at the same configuration or $\sigma \prec_{ca} \sigma'$. In both cases, the construction of W is stopped after σ', hence ρ is not an infinite path, a contradiction.

If the construction of W is completed without finding any two anchored runs such that $\sigma \prec_{ca} \sigma'$ (and then reporting that Sch_{ca} is not regular or that Sch_{ca} is not valid), then $\{(s^\sigma, \chi^\sigma) \mid \sigma \in W\}$ is exactly the set of configurations of $Sch_{ca}(RC)$, hence RC/Sch_{ca} is a finite set, and we can test whether Sch_{ca} is valid by inspecting the finite graph formed of all transitions $(s^\sigma, \chi^\sigma) \xrightarrow{a} (s^{\sigma a}, \chi^{\sigma a})$ in this set. □

Thm. 2 is now settled by applying Prop. 1 and Thm. 4.

Concerning complexity, we rely on coverability techniques which have a non primitive recursive complexity [10] if the order in which paths are explored is not chosen with care. For Petri Nets, [17] explains that if the Karp and Miller tree is generated by exploring paths with the lowest height first, the complexity decreases to doubly exponential time, almost matching the lower bound of [14]. We would need to analyze this or other orders on the paths of our coverability tree to try to obtain elementary complexity. Notice that we cannot even get a lower bound complexity because our systems are not equivalent with Petri Nets.

4.3 Algorithm run on an example

To illustrate the construction given in the proof of Thm. 4, we consider the data-branching system in Fig. 14 and display in Fig. 15 the corresponding tree W constructed in Thm. 4 (for saving space, we contracted `while`,`endfor` and `endwh` as `whl`,`endf` and `endw`). In Fig. 15, the root is indicated by a pointing arrow and each node may be identified by the sequence of labels on the incoming path. We label each node σ with the configuration at which σ ends, and we let the notation $ij\ kl$ represent the configuration in which process $P2$ is at state i, $P1$ is at state j and buffers d,e have respective sizes k,l. A dotted arrow represents a jump to a node already constructed.

Sch_{ca} succeeds to reach the final state $4D00$ through the path (`for while` $!d$ `endfor` $?d$ `endwh` $!e\,?e$) but two deadend states $2D\,01$ and $4D\,10$ may also be reached. This means that Sch_{ca} is not valid, hence by Prop. 1, no valid schedule exists for $\mathcal{A}$. Moreover, (`for endwh` $!d$) $\prec_{ca}$ (`for endwh` $!d$ `for` $!d$). The algorithm thus stops at the configuration $3A\,20$ reached by (`for endwh` $!d$ `for` $!d$),

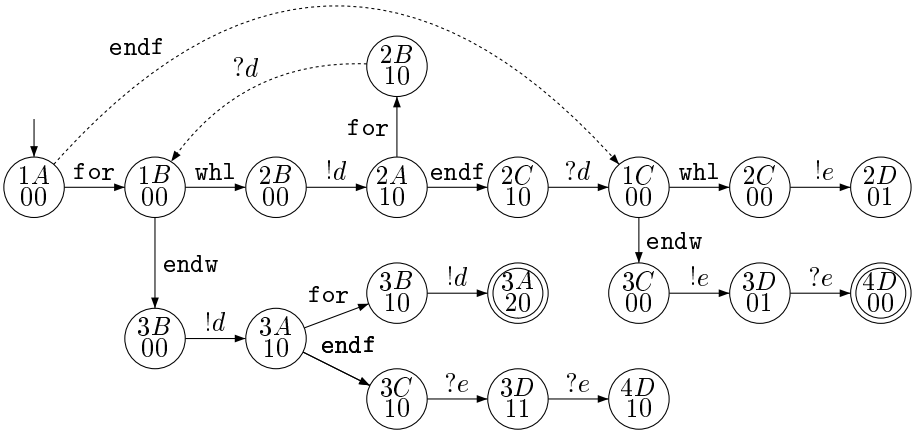

Fig. 15. The tree W in Thm. 4 for the system in Fig. 14.

and depicted by a double circle. This means that the canonical schedule is not regular, and by Prop. 1, no regular and valid schedule exists for $\mathcal{A}$. Notice that the runs (`for endwh`) and (`for while`) are not anchored, hence we do *not* have (`for endwh`) $\prec_{ca}$ (`for endwh` $!d$ `for`). Recall that the model $\mathcal{A}$ is only an over-approximation of the real task system depicted in Fig. 3. That is, even though we have a proof that $\mathcal{A}$ is not schedulable, this does not imply that the real system shown on Fig. 3 is not schedulable.

The algorithm produces counterexample runs (here, we have two runs for deadends and one for unboundedness). One can try to check whether these problematic runs represent actual runs of the original (not abstracted) system or whether they are spurious runs resulting from the abstractions. If all the counterexample runs are spurious, then the real system is in fact schedulable.

5 Discussion

In this paper, we have considered quasi-static scheduling as introduced in [4] and have provided a negative answer to an open question posed in [11]. Specifically we have shown that for the chosen class of infinite state systems, checking whether a system is quasi-static schedulable is undecidable. We have then identified the data-branching restriction, and proved that the quasi-static scheduling problem is decidable for data-branching systems. Further, our proof constructs both the schedule and the finite state behaviour of the system under schedule. An important concept used in the proof is the canonical schedule that draws much inspiration from the study of existential bounds on channels of communicating systems [7]. In the language of [7], our result can be

rephrased as: it is decidable whether a *weak FIFO* data branching communicating system is existentially bounded, when all its local final states are polling states. We recall that the same problem is undecidable [7] for *strong FIFO* communicating systems, even if they are deterministic and deadend free. Our abstraction policy is similar to the one used in [13]. However, we use existential boundedness while [13] checks whether a communicating system is universally bounded, which is an easier notion to check. Note that the canonical schedule may be easily realized in any practical context: it suffices to prevent any process from sending to a buffer that already contains the maximum number of items determined from that schedule. It is also worth recalling that these bounds are optimal (Prop. 1).

Deadends play an important role in the notion of quasi-static schedulability studied here and previously. However, quasi-static scheduling may stumble on spurious deadends due to the modelling of the task by an abstract system. The algorithm we have sketched for constructing the canonical schedule may be combined with an iterative removal of spurious deadends. A more ambitious extension is to design *distributed* quasi-static schedulers for inter cluster communication, where the scheduler cannot have global knowledge of each process.

Acknowledgement: *We would like to thank the reviewers for making constructive comments helping us to improve the readability of the paper.*

References

[1] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. of the ACM*, 30(2):323-342, 1983.

[2] J. Buck. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD Dissertation, Berkeley, 1993.

[3] J. Carmona, J. Cortadella, V. Khomenko and A. Yakovlev. Synthesis of Asynchronous Hardware from Petri Nets. In *Lectures on Concurrency and Petri Nets*, LNCS 3098, pages 345-401, 2003.

[4] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone and Y. Watanabe. Quasi-static scheduling of independent tasks for reactive systems. *IEEE Trans. on Comp.-Aided Design* 24(10):1492-1514, 2005.

[5] Ph. Darondeau, B. Genest, P.S. Thiagarajan and S. Yang. Quasi-Static Scheduling of Communicating Tasks. In *CONCUR 2008*, LNCS 5201, pages 310–324.

[6] D. de Frutos-Escrig. Decidability of home states in place transition systems. Report of Dpto. Informatica y Automatica. Univ. Complutense de Madrid, 1986.

[7] B. Genest, D. Kuske, and A. Muscholl. On Communicating Automata with Bounded Channels. *Fundamenta Informaticae*. 80(2):147–167. 2007.

[8] C. Jard and T. Jeron. Testing for Unboundedness of FIFO Channels. *Theoretical Computer Science*. 113:93–117. 1993.

[9] G. Kahn. The Semantics of Simple Language for Parallel Programming. In *Proc. Int. Federation Information Processing (IFIP) Congress*. pages 471-475. 1974.

[10] R. Karp, R. Miller. Parallel Program Schemata. *J. Comput. Syst. Sci.* 3(2):147-195, 1969.

[11] A. Kondratyev, L. Lavagno, C. Passerone and Y. Watanabe. Quasi-static scheduling of concurrent specifications. In *The Embedded Systems Handbook*, CRC Press, 2005.

[12] M. Sgroi, L. Lavagno, Y. Watanabe and A. Sangiovanni-Vincentelli. Quasi-Static Scheduling of Embedded Software Using Equal Conflict Nets. In *ICATPN 1999*, LNCS 1639, pages 208–227.

[13] S. Leue, R. Mayr and W. Wei. A Scalable Incomplete Test for the Boundedness of UML RT Models. In *TACAS 2004*, LNCS 2988, pages 327–341.

[14] R. Lipton. The Reachability Problem Requires Exponential Space. Research Report 76, Department of Computer Science, Yale University, 1976.

[15] C. Liu, A. Kondratyev, Y. Watanabe, J. Desel, A.L. Sangiovanni-Vincentelli. Schedulability Analysis of Petri Nets Based on Structural Properties. *Fundamenta Informaticae*. 86(3):325–341. 2008.

[16] T. Parks. Bounded Scheduling of Process Networks. PhD Dissertation, EECS Department, Berkeley. 1995.

[17] C. Rackoff. The Covering and Boundedness Problems for Vector Addition Systems. *Theoretical Computer Science* 6:223–231, 1978.